EECS 280 – Lecture 7

Abstract Data Types in C

https://eecs280staff.github.io/notes/07_ADTs_in_C.html

Announcements

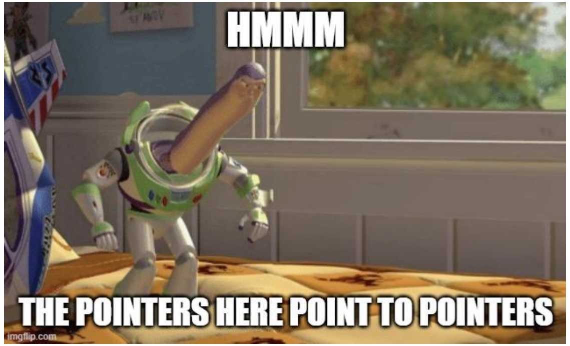
- Lab 3 this week
  - Due Sunday @ 8pm
- Project 2 due Friday @ 8pm
  - See overview session recording on website

Agenda

- **Finish up IO and Streams**
- Abstract Data Types (ADTs) in C
- Representation Invariants
- Testing C-style ADTs

argv and argc

- Two parameters to main:
  - argc – the number of arguments
  - argv – an array of the arguments
- argv is an **array of C-style strings**.

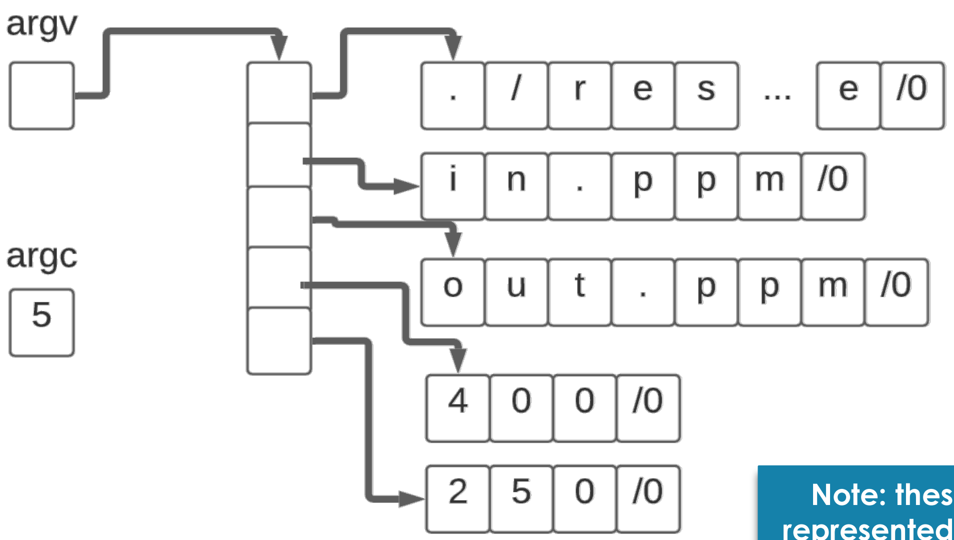



```
int main(int argc, char* argv[]) {

}
```

Compiler turns this into char** argv.

argv and argc

```
$ ./resize.exe in.ppm out.ppm 400 250
```



Poll

- What is the data type of argv[4]?
- What is the value?
- What does `cout << argv[4]` print?

Note: these are all chars represented by ASCII values

atoi

- Arguments are always passed in as c-strings
  - What if we want to treat it like an integer?
    - Like width of PPM file in P2?
  - Doing arithmetic directly on argv[] results in pointer arithmetic… not what we want

```
int main(int argc, char* argv[]) {
  cout << argv[1] << " + 1 = “
       << argv[1] + 1;
}
```

argv[1] is char*, argv[1] + 1 is calculating a new address

```
$ ./program 53
53 + 1 = 0x1001
```

atoi

- Arguments are always passed in as c-strings
  - If we want to treat them like integers, we need to use "atoi"

```
#include <cstdlib>
// EFFECTS: parses s as a number and
//          returns its int value
int atoi(const char* s);
```

```
int main(int argc, char* argv[]) {
  cout << argv[1] << " + 1 = "
       << atoi(argv[1]) + 1;
}
```

```
$ ./program 53
53 + 1 = 54
```

Streams

- An abstraction that allows you to read/write data from input/output

Reading and Writing Images in PPM Format

The `Image` module also provides functions to read and write `Image`s from/to the PPM image format. Here's an example of an `Image` and its representation in PPM.

Image	Image Representation in PPM
	P3 5 5 255 0 0 0 0 0 0 255 255 250 0 0 0 0 0 0 255 255 250 126 66 0 126 66 0 126 66 0 255 255 250 126 66 0 0 0 0 255 219 183 0 0 0 126 66 0 255 219 183 255 219 183 0 0 0 255 219 183 255 219 183 255 219 183 0 0 0 134 0 0 0 0 0 255 219 183

Streams

- Example input streams:
  - Standard input
    - a.k.a “cin”
    - reads from terminal
  - ifstream
    - input file stream
    - read from a file
- Read using “extraction operator” >>

- Example output streams:
  - Standard output
    - “cout”
    - writes to terminal
  - ofstream
    - output file stream
    - write to a file
- Write using “insertion operator” <<

cin Example

- We're already familiar with reading input from standard input (`cin`).

words.cpp

```
string word;
while (cin >> word) {
  cout << "word = '" << word << "'" << endl;
}
```

Will stop when an "end of file" character is read. To type this at the console, use ctrl+d.

```
$ g++ words.cpp -o words

$ ./words
hello world!
word = 'hello'
word = 'world!'
goodbye
word = 'goodbye'
```

File I/O with Streams

- In C++, we can read and write files directly with `ifstream` and `ofstream` objects

```
#include <fstream>
```

- `ifstream` and `ofstream` allow you to…
  - …read a file just like reading from `cin`
  - …write to a file just like printing to `cout`

File Input: `ifstream`

```
int main() {
  string filename = "hello.txt";
  ifstream fin;
  fin.open(filename);
  if (!fin.is_open()) {
    cout << "open failed" << endl;
    return 1;
  }
  string word;
  while (fin >> word) {
    cout << "word = '" << word << "'" << endl;
  }
  fin.close();
}
```

Open a file using `fin` variable

Check for success opening file.

Read one word at a time and check that the read was successful.

File Output: `ofstream`

```
int main() {
  const int SIZE = 4;
  int data[SIZE] = { 1, 2, 3, 4 };
  string filename = "output.txt";
  ofstream fout;
  fout.open(filename);
  if (!fout.is_open()) {
    cout << "open failed" << endl;
    return 1;
  }
  for (int i = 0; i < 4; ++i) {
    fout << "data[" << i << "] = " << data[i] << endl;
  }
  fout.close();
}
```

output.txt

```
data[0] = 1
data[1] = 2
data[2] = 3
data[3] = 4
```

Using istream/ostream generically

- ofstream instances and cout are both instances of the more generic “ostream”
  - And ifstream instances and cin are both instances of “istream”
  - We can write generic functions that work for either (used in P2!)

```
void print(ostream& os) {
  os << "hi" << endl;
}
int main() {
  ofstream fout;
  fout.open("output.txt");
  print(fout);  // prints to output.txt
  print(cout);  // prints to terminal
}
```

This function doesn't know/care whether it's printing to cout, a file, or whatever

Motivation

- Running tests in P1 wasn't too bad
  1. Initialize vector with data
  2. Pass vector to function
  3. Assert that result is equal to the correct value
- But in P2 we need to deal with files!
  - We could check the files afterwards with "diff"
  - … but having a separate file for each test will be cumbersome
  - And the autograder doesn't support running "diff"
- Solution: **stringstreams**

Stringstreams and Testing

- `istringstream`
  - An input stream that uses a `string` as its source.
  - Useful for simulating stream input from a "hardcoded" `string`.

```
TEST(test_image_basic) {
  // A hardcoded PPM image
  string input = "P3\n2 2\n255\n255 0 0 0 255 0 \n";
  input += "0 0 255 255 255 255 \n";

  // Use istringstream for simulated input
  istringstream ss_input(input);
  Image* img = new Image;
  Image_init(img, ss_input);

  ASSERT_EQUAL(Image_width(img), 2);
  Pixel red = { 255, 0, 0 };
  ASSERT_TRUE(Pixel_equal(Image_get_pixel(img, 0, 0), red));
  delete img;
}
```

Stringstreams and Testing

- `ostringstream`
  - An output stream that writes into a `string`.
  - Useful for capturing output as a `string` that can be checked for correctness.

```
TEST(test_matrix_basic) {
  Matrix* mat = new Matrix;
  Matrix_init(mat, 3, 3);
  Matrix_fill(mat, 0);
  Matrix_fill_border(mat, 1);

  // Hardcoded correct output
  string output_correct = "3 3\n1 1 1 \n1 0 1 \n1 1 1 \n";

  // Capture output in ostringstream
  ostringstream ss_output;
  Matrix_print(mat, ss_output);
  ASSERT_EQUAL(ss_output.str(), output_correct);
  delete mat;
}
```

Agenda

- Finish up IO and Streams
- **Abstract Data Types (ADTs) in C**
- Representation Invariants
- Testing C-style ADTs

Motivation

- Abstraction: *Removing certain details to focus attention on other, more important details*
  - Functional abstraction: presenting complex tasks as function calls
  - Data abstraction: presenting complex data types as a set of composite objects and functions to interact with those objects

Functional Abstraction

- Reality: complex code sequences (e.g. extract_column implementation)
- Abstraction: single function call defined by input parameters and return value (e.g. extract_column function call)

```
std::vector<double> extract_column(std::string filename,
                                   std::string column_name) {

  // open file
  ifstream fin;
  fin.open(filename.c_str());
  if (!fin.is_op       {
    cout << "E         " << filena
    exit(1)
  }

  // use csvstream
  csvstream csvin(fi

  // check fo
  vector<s            .g
  size_t colu         ze();
  for (size_t i        header.size();
    if (header[i]    column_name) {
      column =
      break;
    }
  }
```

Hide from programmer in p1_library.cpp

```
//EFFECTS: extracts one column of data from a tab separated values file (.tsv)
//  Prints errors to stdout and exits with non-zero status on errors
std::vector<double> extract_column(std::string filename, std::string column_name);
```

Present to programmer in p1_library.h

Data Abstraction

- Reality: several data variables organized in unintuitive ways
  - E.g. a 2d image represented as a long 1D array
- Abstraction: An Abstract Data Type (ADT)
  - Hides the data representation behind a composite type (e.g. struct) and functions that interact with it
  - E.g. the “Matrix” data type in P2

```
// Representation of a 2D matrix of integers
// Matrix objects may be copied.
struct Matrix{
  int width;
  int    ight;
  int data[          MAX_MATRIX_HEIGHT];
};
```

Hide from programmer

```
// EFFECTS:  Returns a pointer to the element in the Matrix
//           at the given row and column.
int* Matrix_at(Matrix* mat, int row, int column);
```

Present to programmer in Matrix.h

Abstract Data Types (ADTs)

- Again, separate **interface** from **implementation**.
- Example, C++ strings:

```
string str1 = "hello";
string str2 = "jello";
cout << str1 << endl;
if (str1.length() == str2.length()) {
  cout << "Same length!" << endl;
}
```

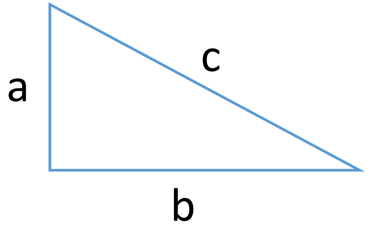
- Unlike C-style strings, we don't need to understand the inner workings of the "string" type in order to use it
- See also: vectors used in project 1

Benefits of ADTs

Poll: What is the advantage of using ADTs? (select all that apply)

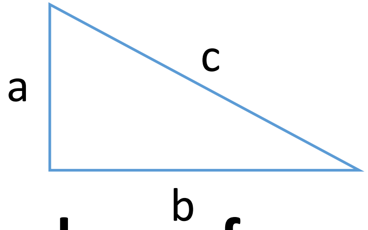
A) Code is easier to read
B) Code is easier to update with changes
C) Code is faster when compiled
D) It’s a great conversation piece on dates

C-Style ADTs



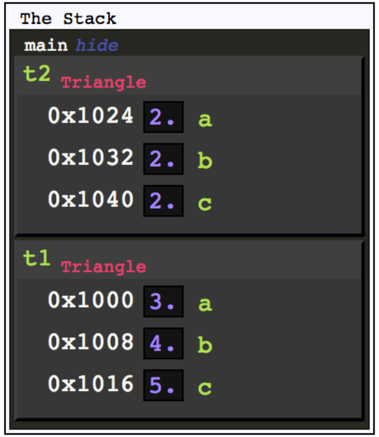
- Let's say we want to represent **triangles**
- First, pick a **data representation**
  - We have many choices!
  - Could use three side lengths, three angles, three sets of coordinates… etc
  - This is an **implementation detail**, we want to hide this information from the user

C-Style ADTs



- Let's choose three side lengths **a**, **b**, **c**, as **members of a struct** for our representation

```
struct Triangle {
  double a;
  double b;
  double c;
};

int main() {
  Triangle t1;
  Triangle t2;
  // set values for t1 and t2
}
```



C-Style ADTs (structs)

But why?! Isn't this just more code?

- Rather than directly access member variables **directly** in our code, define a set of functions (the "**interface**") to interact with these objects **indirectly**

```
struct Triangle {
  double a;
  double b;
  double c;
};

int main() {
  Triangle t1 = { 3, 4, 5 };
  // print perimeter
  cout << t1.a + t1.b + t1.c;
}
```

Bad - don't want to do this

```
//Triangle.h
struct Triangle {
  double a;
  double b;
  double c;
};

void Triangle_init(Triangle* tri,
    double a, double b, double c) {
  tri->a = a;
  tri->b = b;
  tri->c = c;
}

double Triangle_perimeter(
      const Triangle* tri) {
  return tri->a + tri->b + tri->c;
}
```

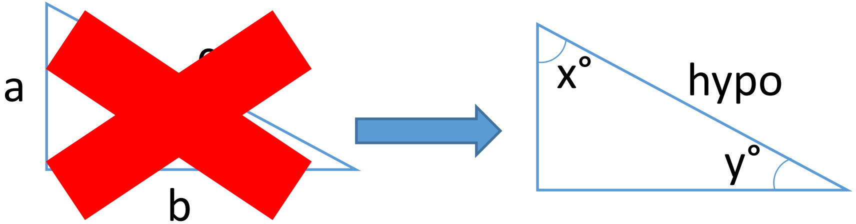
```
#include "Triangle.h"

int main() {
  Triangle t1;
  Triangle_init(&t1 3, 4, 5 };
  cout<<Triangle_perimeter(&t1);
}
```

Good!

Scenario

```
struct Triangle {
  double angle_x;
  double angle_y;
  double hypo;
};
```

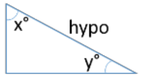
- Your project lead gives you specification to store triangle’s lengths of lines
- You work for a month, write 1000s of lines using your Triangle struct
- Then… call comes in to change representation!
  - Maybe they determined storing angles would be more efficient int the long run



Scenario

```
str  t Triangl  {
          a;
  do
  do
};
```

```
struct Triangle {
  double angle_x;
  double angle_y;
  double hypo;
};
```



- All your old code needs to be modified!
- Unless you respected your interface!!
  - Then you only need to update the functions implementations in 1 file

```
void Triangle_init(Triangle* tri,
    double a, double b, double c) {
  tri->angle_x = asin(b/c);
  tri->angle_y = asin(a/c);
  tri->hypo = c;
}
```

Update this once, then all uses should be fine

```
int main() {
  Triangle t1 = { 3, 4, 5 };
  cout << t1.a + t1.b + t1.c;
}
```

"Breaking the interface" – results in 1000s lines of code broken

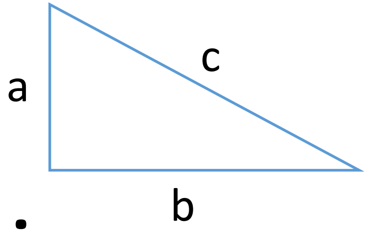
```
#include "Triangle.h"

int main() {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 5 };
  cout<<Triangle_perimeter(&t1);
}
```

Works fine once "Triangle_init" & rest of "Triangle.cpp" is updated

Takeaway

- Hiding the **implementation details** of a data structure behind a set of **well defined functions** (i.e. an **interface**) makes it much easier to build scalable code
  - Easier to change
  - Easier to understand
- **Interface** - the "what" - defined in .h file
- **Implementation** - the "how" - implemented in .cpp file

C-Style ADTs (structs)



- Define functions for `Triangle` **behaviors**.
- These determine the **interface** for a triangle

```
struct Triangle {
  double a, b, c;
};

double Triangle_perimeter(const Triangle* tri) {
  return tri->a + tri->b + tri->c;
}



int main() {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 5 };
  cout<<Triangle_perimeter(&t1);
}
```

The first parameter is a pointer to the `Triangle` struct we want to work with.

```c
struct Triangle {
  double a, b, c;
};
```

Let's say we want to add a function to scale triangles by a given factor.

Poll:

Which of these are correct? (select all that apply)

A

```c
void Triangle_scale(const Triangle* tri,
                    double s) {
  tri->a *= s;
  tri->b *= s;
  tri->c *= s;
}
```

B

```c
void Triangle_scale(Triangle tri,
                    double s) {
  tri.a *= s;
  tri.b *= s;
  tri.c *= s;
}
```

C

```c
void Triangle_scale(Triangle* tri,
                    double s) {
  a *= s;
  b *= s;
  c *= s;
}
```

D

```c
void Triangle_scale(Triangle* tri,
                    double s) {
  tri->a *= s;
  tri->b *= s;
  tri->c *= s;
}
```

E

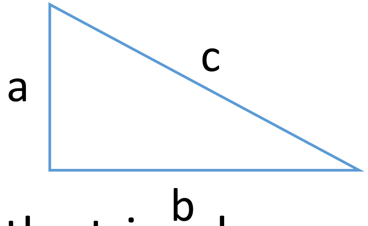
```c
void Triangle_scale(double s) {
  t1.a *= s;
  t1.b *= s;
  t1.c *= s;
}
```

x *= y is shorthand for x = x*y (just like +=)

Agenda

- Finish up IO and Streams
- Abstract Data Types (ADTs) in C
- **Representation Invariants**
- Testing C-style ADTs

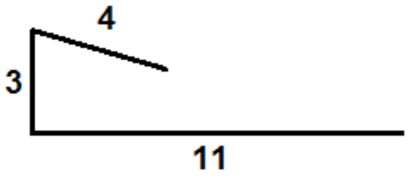
C-Style ADTs (structs)



- There are some issues with the way we're initializing the triangle.
- What's wrong with this code?

```
int main() {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 11 );
  Triangle_scale(&t1, 2);
  cout << Triangle_perimeter(&t1) << endl;

}
```

We have no check on the values used to initialize the `Triangle` member variables.



Representation Invariants

- A problem for compound types…
  - Some combinations of member values don't make sense together.
- We use **representation invariants** to express the conditions for a **valid** compound object.
- "REQUIRES for ADTs"
- For Triangle:

Positive Edge Lengths

```
0 < a
0 < b
0 < c
```

Triangle Inequality

```
a + b > c
a + c > b
b + c > a
```

C-Style ADTs (`structs`)

- Solution: Define an initializer function, check invariants with assert

```
struct Triangle {
  double a, b, c;
};

void Triangle_init(Triangle* tri, double a_in,
                   double b_in, double c_in) {
  assert(0 < a_in && 0 < b_in && 0 < c_in);
  assert(a_in + b_in > c_in  && a_in + c_in > b_in &&
         b_in + c_in > a_in);
  tri->a = a_in;
  tri->b = b_in;
  tri->c = c_in;
}

int main() {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 11);
}
```

This will now cause a failed assertion.

Abstraction Layers

- ADTs can be composed to provide multiple layers of abstraction.

Image "what".

	0	1	2	3	4
0	(0,0,0)	(0,0,0)	(255,255,250)	(0,0,0)	(0,0,0)
1	(255,255,250)	(126,66,0)	(126,66,0)	(126,66,0)	(255,255,250)
2	(126,66,0)	(0,0,0)	(255,219,183)	(0,0,0)	(126,66,0)
3	(255,219,183)	(255,219,183)	(0,0,0)	(255,219,183)	(255,219,183)
4	(255,219,183)	(0,0,0)	(134,0,0)	(0,0,0)	(255,219,183)

Image "how", using Matrix "what".

	0	1	2	3	4
0	0	0	255	0	0
1	255	126	126	126	255
2	126	0	255	0	126
3	255	255	0	255	255
4	255	0	134	0	255

	0	1	2	3	4
0	0	0	255	0	0
1	255	66	66	66	255
2	66	0	219	0	66
3	219	219	0	219	219
4	219	0	0	0	219

	0	1	2	3	4
0	0	0	250	0	0
1	250	0	0	0	250
2	0	0	183	0	0
3	183	183	0	183	183
4	183	0	0	0	183

Matrix "how".

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0	0	255	0	0	255	126	126	126	255	126	0	255	0	126	255	255	0	255	255	255	0	134	0	255

Abstraction Layers

- In P2, when building "resize.cpp" we don't care that Image consists of a Matrix, which contains 3 1-D arrays of pixel values
  - We just rely on the abstraction functioning correctly and focus on our current "layer"

```
Image i;
Image_init(&i, file );
my_width = Image_width(&i);
```

Agenda

- Finish up IO and Streams
- Abstract Data Types (ADTs) in C
- Representation Invariants
- **Testing C-style ADTs**

Kinds of Test Cases

Consider test cases for the `Matrix_at` function from project 2...

```
// REQUIRES: mat points to a valid Matrix
//           0 <= row && row < Matrix_height(mat)
//           0 <= column && column < Matrix_width(mat)
// EFFECTS:  Returns a pointer to the element in the Matrix
//           at the given row and column.
int* Matrix_at(Matrix* mat, int row, int column);
```

REQUIRES Prohibited	ASSERT_EQUAL(*Matrix_at($\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$, 1, -1), 42)
Simple	ASSERT_EQUAL(*Matrix_at($\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$, 1, 1), 5)
(Edge) Special	ASSERT_EQUAL(*Matrix_at($\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 9\end{bmatrix}$, 2, 2), 9)
Stress	Matrix_init(big, 400, 400); Matrix_fill(big, 1); ASSERT_TRUE(Matrix_equal(big, $\begin{bmatrix}1 & \cdots & 1\\\vdots & \ddots & \vdots\\1 & \cdots & 1\end{bmatrix}$));

Don't write this

Not needed for P2.

Respect the Interface?

- Is it okay for tests to “break” the interface?

```
TEST(perimeter) {
  Triangle t1;
  Triangle_init(&t1, 3, 4, 5);
  int actual = Triangle_perimeter(&t1);
  // is this okay?
  int expected = t1.a + t1.b + t1.c;
  ASSERT_EQUAL(actual, expected);
}
```

```
struct Triangle {
  double a;
  double b;
  double c;
};
```

A) No – because you told me
B) No - because our tests will break if we do and then change the implementation
C) Yes - tests are for ourselves and therefore don't need to be as "polished"
D) Yes - because I'm an adult and I do as I please

Simple Test

```
// Fills a 3x5 Matrix with a value and checks that
// Matrix_at returns that value for each element.
TEST(test_fill_basic) {
  Matrix* mat = new Matrix;
  const int width = 3;
  const int height = 5;
  const int value = 42;
  Matrix_init(mat, width, height);
  Matrix_fill(mat, value);
  for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
      ASSERT_EQUAL(*Matrix_at(mat, row, col), value);
    }
  }
  delete mat;
}
```

Bad Edge Test

```
// Places the maximum value at a corner of the
// matrix and tests that Matrix_max finds it.
TEST(edge_test_max) {
  Matrix* mat = new Matrix;
  const int width = 3;
  const int height = 5;

  Matrix_init(mat, width, height);
  for (int i = 0; i < width * height; ++i) {
    mat->data[i] = i;
  }

  mat->data[14] = 99;

  ASSERT_EQUAL(Matrix_max(mat), 99);
  delete mat;
}
```

Breaks the Matrix interface.

Good Edge Test

```
// Places the maximum value at a corner of the
// matrix and tests that Matrix_max finds it.
TEST(edge_test_max) {
  Matrix* mat = new Matrix;
  const int width = 3;
  const int height = 5;
  const int max_value = 99;
  Matrix_init(mat, width, height);
  for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
      *Matrix_at(mat, row, col) = row * width + col;
    }
  }
  *Matrix_at(mat, 4, 2) = max_value;
  ASSERT_EQUAL(Matrix_max(mat), max_value);
  delete mat;
}
```

Don't worry about new/delete right now

Often times with data structures, literally checking "the edges" is a good edge test

Next Time

- Abstract data types in C++
  - What extra features "classes" give us

- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah